Teoría de Lenguajes - Trabajo Práctico
Analizador Sintáctico y Semántico para λ^{bn}

Versión 1.1

1er cuatrimestre 2017

Fecha de entrega: miércoles 5 de julio

1. Introducción

Se desea crear un analizador sintáctico y semántico para un subconjunto del cálculo lambda tipado, sobre booleanos y naturales (λ^{bn}). Así, se deberá analizar si una expresión sigue la sintaxis esperada, y, en caso de que sea válida, evaluarla adecuadamente e indicar de qué tipo es la misma. Para la evaluación se puede tomar como referencia la semántica operacional descripta en la 1era clase teórica y práctica de la materia Paradigmas de Lenguajes de Programación[1], aunque no es necesario seguir el proceso de evaluación paso a paso.

2. Descripción del lenguaje de entrada

A efectos de este trabajo, el cálculo lambda tipado sobre booleanos y naturales deberá soportar los siguientes términos:

```
M  ::=  x  | true | false | if M1 then M2 else M3 | \x:T.M | M1 M2 | 0 |
        succ(M) | pred(M) | iszero(M)
```

Siendo,

- `true` y `false`: las constantes de verdad,
- `if M1 then M2 else M3`: el condicional, por lo que si `M1` se evalua a `true`, resulta en `M1` y, si no, en `M2`, y tanto `M1` como `M2` deben ser del mismo tipo,
- `M1 M2`: la aplicación de la función denotada por el término `M1` al argumento `M2`, por ejemplo, `(\x:Nat.succ(succ(x))) 3`, se evalúa a 5, siendo este operador asociativo a izquierda (i.e., `M N P` se evalúa como `(M N) P`),
- `\x:T.M`: una función anónima cuyo parámetro formal es `x`, de tipo `T`, y tiene a `M` como cuerpo, siendo la operación de menor precedencia,
- `x`: una variable de términos, que puede estar asociada a cualquier término válido,
- `succ(M)`: el término para el cual se evaluará M hasta obtener un número, y se lo incrementará,
- `pred(M)`: el término para el cual se evaluará M hasta obtener un número, y se lo decrementará,
- `iszero(M)`: el término para el cual se evaluará M hasta obtener un número, y evaluará a `true` o `false` según sea cero o no.

A su vez, cada expresión bien formada del lenguaje puede tomar alguno de los siguientes tipos:

```
T  ::=  Bool | Nat | T1 -> T2
```

Siendo,

- `Bool`: el tipo para los booleanos (e.g., `isZero(0): Bool`),
- `Nat`: el tipo para los naturales, (e.g., `succ(pred(0)): Bool`),
- `T1 -> T2`: el tipo para las funciones que van de `T1` a `T2`,
  por ejemplo, `\x:Bool.if x then succ(0) else pred(succ(succ(0))): Bool -> Nat`

Una expresión del cálculo lambda está en forma normal si no puede evaluarse más. Así, los valores están en forma normal, aunque no todops los términos que no pueden evaluarse más son valores. Finalmente, si tenemos un término cerrado (i.e., que no tiene variables libres) y que está bien tipado, podremos obtener un valor a partir de él.

Los valores para nuestro lenguaje serán los siguientes:

```
V  ::=  true | false | \x:T.M | n
```

con `n` como macro de $\texttt{succ}^n\texttt{(0)}$

3. Descripción del lenguaje de salida

Dada una cadena de entrada, se deberá evaluar si efectivamente respeta la sintaxis esperada de Cálculo Lambda y, si no hay algún de error de tipado[1], se deberá evaluar la expresión hasta llegar a un valor, e indicar el tipo del mismo.

3.1. Ejemplos

Entrada	Resultado	Salida
`0`	OK	`0:Nat`
`true`	OK	`true:Bool`
`if true then 0 else false`	ERROR: Las dos opciones del if deben tener el mismo tipo	
`\x:Bool.if x then false else true`	OK	`\x:Bool.if x then false else true:Bool->Bool`
`\x:Nat.succ(0)`	OK	`\x:Nat.succ(0):Nat->Nat`
`\z:Nat.z`	OK	`\z:Nat.z:Nat->Nat`
`(\x:Bool.succ(x)) true`	ERROR: succ espera un valor de tipo Nat	
`succ(succ(succ(0)))`	OK	`succ(succ(succ(0))):Nat`
`x`	ERROR: El término no es cerrado (x está libre)	
`succ(succ(pred(0)))`	OK	`succ(succ(0)):Nat`
`\x:Nat.succ(x)`	OK	`\x:Nat.succ(x):Nat->Nat`
`0 0`	ERROR: La parte izquierda de la aplicación (0) no es una función con dominio en Nat	
`\x:Nat->Nat.\y:Nat. (\z:Bool.if z then x y else 0)`	OK	`\x:Nat->Nat.\y:Nat.(\z:Bool.if z then x y else 0): (Nat->Nat)->(Nat->(Bool->Nat))`
`(\x:Nat->Nat.\y:Nat. (\z:Bool.if z then x y else 0)) (\j:Nat.succ(j)) 8 true`	OK	`9:Nat`

[1] Esto es, no se corresponde el tipo esperado con el recibido

4. Modo de uso

El programa deberá poder ejecutarse como un comando de consola del sistema operativo, con los siguientes requerimientos:

Sinopsis:

./CLambda [EXPRESION]

Descripción:

El programa deberá recibir una cadena con la expresión a evaluar (EXPRESION), y deberá devolver por *standard output* (`stdout`) el resultado de la evaluación.
Si no se especificara la cadena en la llamada, se esperará recibirla por *standard input* (`stdin`). En caso de que hubiera algún inconveniente al ejecutar el programa, se deberá terminar el programa con código de salida de error (esto es, un código mayor a 0), y mostrar los detalles por standard error (`stderr`).
En caso de que la llamada fuera correcta, se deberá devolver el resultado, y, si no fuera correcta la expresión, incluir todos los detalles del error por `stderr` y no retornar nada por `stdout`.

5. Implementación

Para el lenguaje descripto se solicita crear un analizador léxico, generando los tokens necesarios de acuerdo a la gramática que se haya diseñado, y un analizador sintáctico, que deberá generar el árbol sintáctico para la cadena procesada, en caso de que sea válida, o un mensaje de error adecuado, en caso de que no lo sea. Para este último caso, será deseable indicar el lugar preciso en el cual se encuentra el problema detectado.

Hay dos grupos de herramientas que se pueden utilizar para generar los analizadores léxicos y sintácticos:

- uno utiliza expresiones regulares y autómatas finitos para el análisis lexicografico, y la técnica LALR para el análisis sintáctico. Ejemplos de esto son lex y yacc, que generan código C o C++, JLex y CUP, que generan código Java y ply, que genera código Python. flex y bison son implementaciones libres y gratuitas de lex y yacc.
  En particular, permitiremos utilizar PLY 3.6[2] (o superior).
- el otro grupo utiliza la técnica ELL(k), tanto para el análisis léxico como para el sintáctico, generando parsers descendentes iterativos recursivos. Ejemplos son JavaCC y ANTLR, que están escritos en Java. JavaCC puede generar código Java o C++. ANTLR puede generar Java, C#, Python y JavaScript.
  En particular, para este trabajo se podrá usar ANTLR 4.X[3], y se recomienda utilizar el plugin[4] para Eclipse.

6. Detalles de la entrega

Se deberá enviar el código a la dirección de e-mail tptleng@gmail.com, satisfaciendo lo siguiente:

- el **asunto de mail** debe ser `[TL-TP]` seguido por el nombre del grupo (e.g., "[TL-TP] The Chomsky Boys").
- en el mail deberán estar copiados todos los/as integrantes del grupo.

[2] PLY: Python Lex & Yacc. Disponible en https://pypi.python.org/pypi/ply
[3] ANTLR (ANother Tool for Language Recognition): http://www.antlr.org
[4] ANTLR plugin for Eclipse: http://antlreclipse.sourceforge.net

La entrega debe incluir lo descripto a continuación:

- un programa que cumpla con lo solicitado,
- el código fuente del mismo, adecuadamente documentado,
- informe enviado por e-mail y entregado impreso, con los siguientes contenidos:
  - carátula con datos de integrantes del grupo y nombre del grupo,
  - breve introducción al problema a resolver,
  - los tokens, con sus expresiones regulares, y la gramática no ambigua definidos a partir del lenguaje propuesto,
  - indicación del tipo de la gramática definida, de acuerdo a los vistos en clase,
  - el código de la solución, y, si se usaron herramientas generadoras de código, imprimir la fuente ingresada a la herramienta, no el código generado,
  - descripción de cómo se implementó la solución, con decisiones que hayan tenido que tomar y justificación de las mismas,
  - información y requerimientos de software para ejecutar y recompilar el tp (versiones de compiladores, herramientas, plataforma, etc), como un pequeño manual del usuario, que además de los requerimientos, contenga instrucciones para compilarlo, ejectuarlo, información de parámetros y lo que consideren necesario,
  - casos de prueba con expresiones sintácticamente correctas e incorrectas (al menos tres para cada caso) y
  - un resumen de los resultados obtenidos, y conclusiones del trabajo.

Es parte de lo que se espera de la resolución del trabajo práctico la detección de puntos no especificados en el enunciado y su resolución. En cualquier caso, siempre pueden realizar consultas al respecto.

Referencias

[1] Descargas de la materia Paradigmas de Lenguajes de Programación, 1er cuatrimestre de 2017. Disponible en http://www.dc.uba.ar/materias/plp/cursos/2017/cuat1/descargas/.

[2] Aho, A.V., Lam, M.S., Sethi, R., *Compilers: Principles, Techniques and Tools - Second Edition*, Pearson Education, 2007.

[3] Goyvaerts, J., Levithan, S., *Regular Expressions Cookbook*, O'Reilly, 2009.

[4] Grune, D., Jacobs, C.J.H, *Parsing Techniques: A Practical Guide - Second Edition*, Springer, 2008.

[5] Hopcroft, J.E., Motwani, R., Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation - Third Edition*, Addison Wesley, 2007.

[6] Parr, T., *The Definitive ANTLR 4 Reference*, The Pragmatic Programmers, 2012.

[7] PLY (Python Lex-Yacc). Documentación. Disponible en http://www.dabeaz.com/ply/ply.html